CS190C Lec3

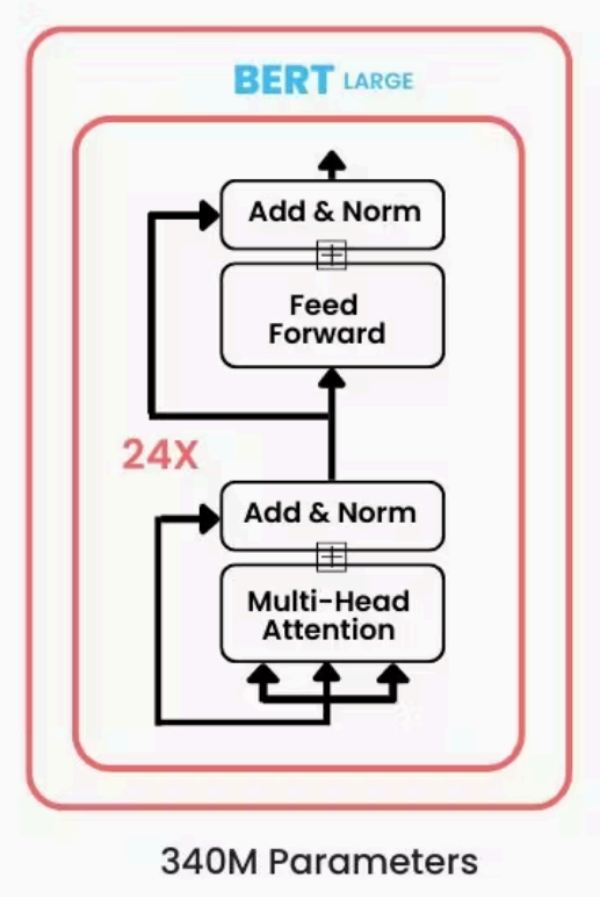
Iterations of Transformer

Overview

- Architecture
- Attention
  - KV Cache
  - RoPE
  - Sliding window/Sparse attention
- FFN
  - SwiGLU
  - MoE
- Normalization
  - Normalization Position
  - Normalization Methods

PART1: Architecture

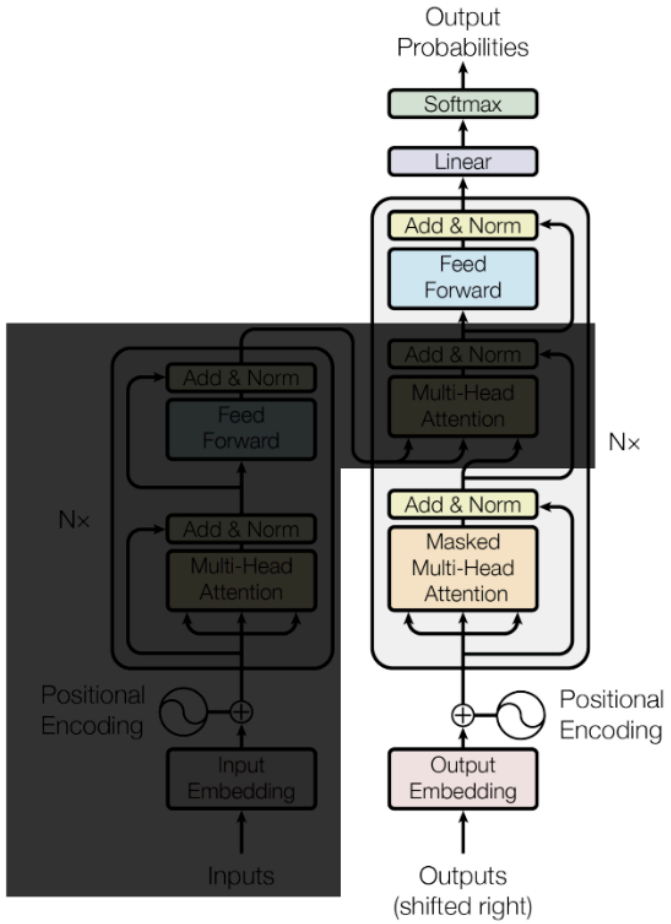
1.Encoder model

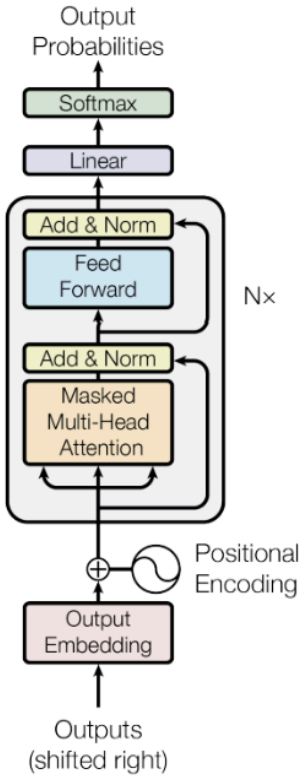


One of encoder models is BERT. It is actually the encoder part of Original Transformer. https://arxiv.org/abs/2104.09864

- **What can it do**: Given a masked sentence `Andrew Ng [MASK] an ML researcher.` It can infer the embedding of all `[MASK]` s.
- **How to train it**: `MLM` (masked language model) method: given sentences with masks at a certain ratio. Compare its output with standard answer for `[MASK]`
- Good at tasks about understanding the text, but poor in generating.

2.Decoder model



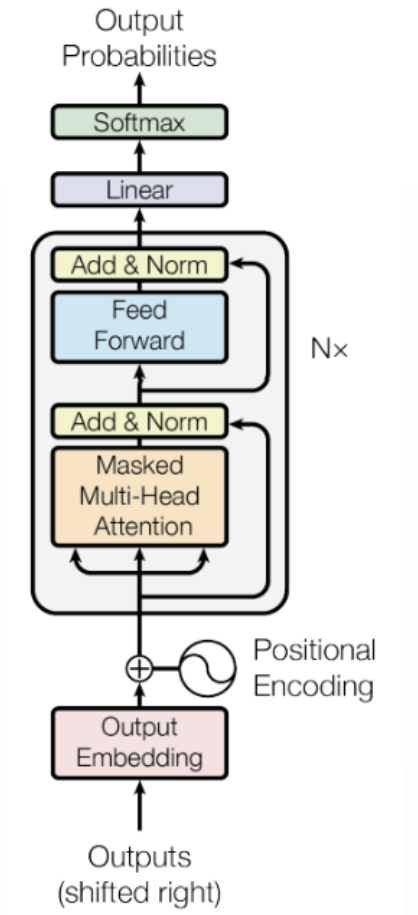


One of decoder models is GPT-3.
https://arxiv.org/abs/2005.14165

It is the right part of the image:

- Remove encoder of Original Transformer.
- Remove cross-attention of Original Transformer.

2.Decoder model



- **What can it do**: Given a sentence, and it can autoregressively decode:
  - Predict next token based on the given sentence now.
  - Add predicted token to "given sentence" then.
- **How to train it**: `CLM` (causal language model) method: given an input sentence and an output sentence. Compare the output of model based on input sentence with the standard output sentence. (More in Lec 6)
- Good at generating. But perform not as good as Encoder for understanding tasks. (Only unidirectional attention)

3.Encoder-Decoder model

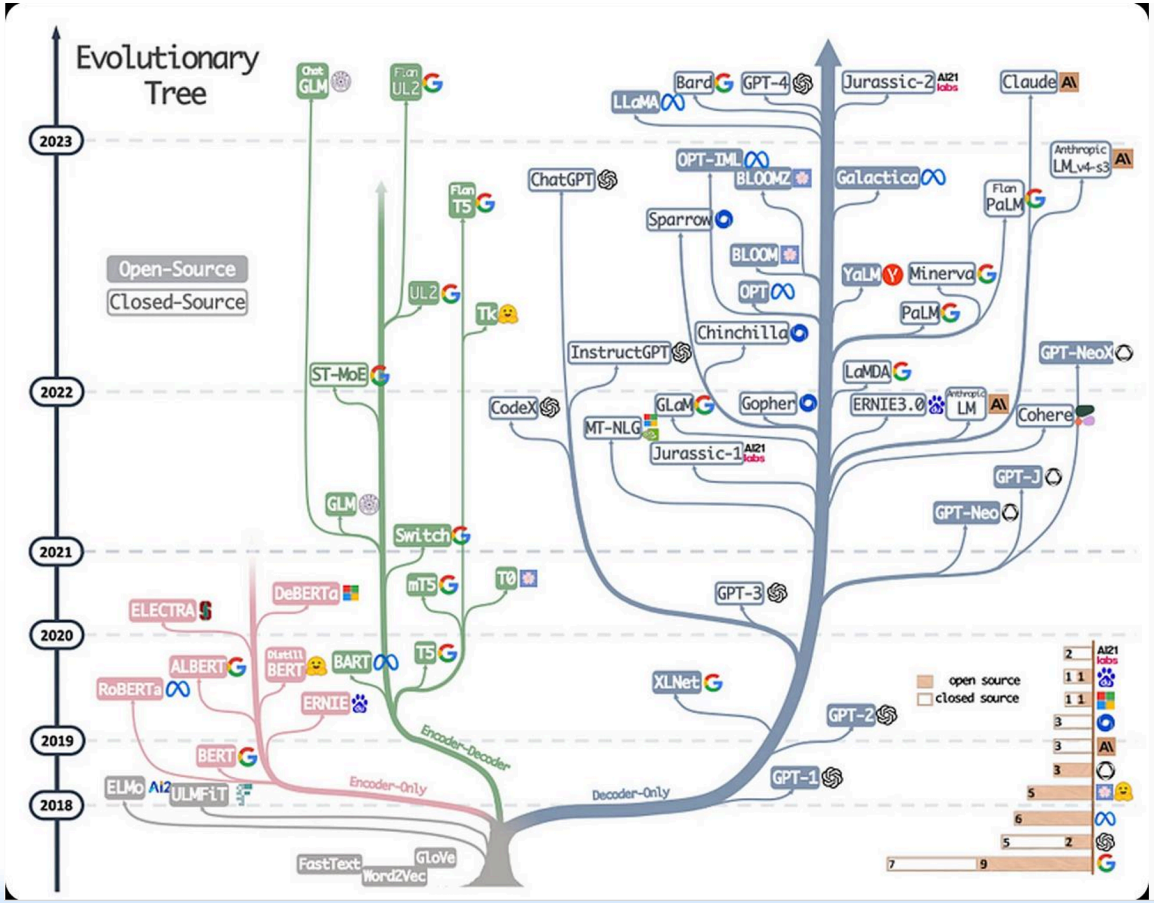
Similar to Original Transformer. One of this kind of model is T5.
https://arxiv.org/abs/1910.10683

Good at tasks like Seq2Seq. But this kind of models are now almost completely replaced by Decoder models.

- Training method: mask a part of sentence and try to fix it. It is not similar to "prompt+answer" format (infer task), need further finetuning.
- For certain parameter amount, a considerable part lies in encoder. But we only need one time of encoding, and `seq_len` times of decoding. Wasted.
- Should cache multiple groups of KVs, especially cross-attention is more complex to maintain. Cost significantly more memory.

For architecture……

We will focus more on Decoder model in this course, and it is truly the most common type of architecture nowadays.



PART2: Attention

PART2.1: RoPE

https://arxiv.org/abs/2104.09864

Position embedding

Original Transformer use sine position encoding, but as we discussed in Lec2, it has several disadvantages, especially the problem of absolute position index and problems of very long sentences in inference cause by it.

Most LLMs nowadays use RoPE as position embedding method.

RoPE

$\vec{x} \Rightarrow R_i\vec{x}$ ($\vec{x} \in R^d$,d is even)

Divide the d-dimensional vector into several sub-segments of length 2, resulting in a total of $d/2$ sub-segments, each sub-segment makes a rotation of angle $\theta_{i,k}$.
(Proportional to position i)

$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \qquad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

where $\theta_{i,k} = \frac{i}{\Theta^{2k/d}}$

RoPE

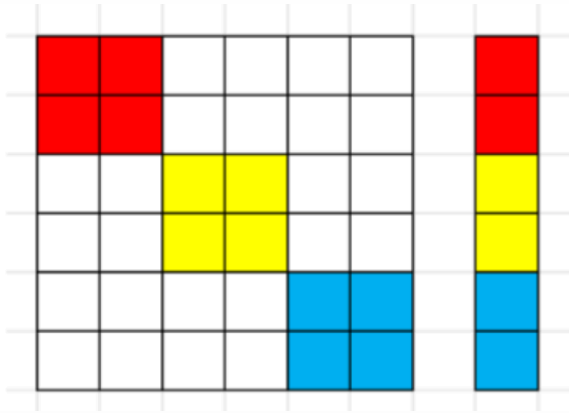
$$R^i = \begin{bmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{bmatrix} \quad R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}.$$

The transformation caused by position, is only related to relative position.

Property of matrix R : $R^m(R^n)^T = R^{m-n}$ (Transpose is inverse, multiply is add in power)

In the attention mechanism, for the query vector q_i at position i and the key vector k_j at position j :

- $q_i' = R^i q_i,\ k_j' = R^j k_j$
- $(q_i')^T k_j' = q_i^T (R^i)^T (R^j) k_j^T = q_i^T R^{j-i} k_j$

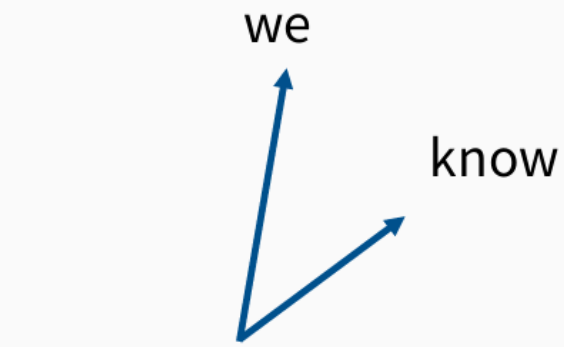


How to understand it?

$$(q_i')^T k_j' = q_i^T (R^i)^T (R^j) k_j^T = q_i^T R^{j-i} k_j$$

- Rotate p and q with their absolute position, and pure relative rotation without any other transformation (like addition) causes relative position.
  - Review: Sine method has absolute position related terms.
- Only change direction, do not change length.
  - $\|q^T k\| = \|q\| \cdot \|k\| \cos(\Delta\theta) \Rightarrow \text{Attention logit}$
  - If we use sine position encoding method, the influence of position embedding to length of vector is existent and uncontrollable, causing distortion in attention logits.
  - But in RoPE, same relative position and two vectors must cause the same attention logits!

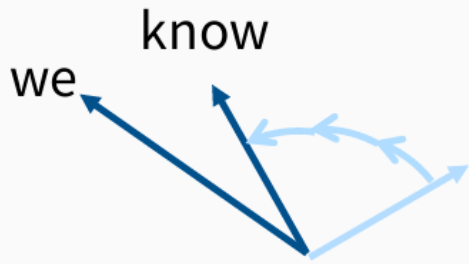
How to understand it?



Position independent embedding

Embedding
“we know that”

Rotate we by ‘0 positions’
know by ‘1 positions’



Embedding
“of course we know”

Rotate we by ‘2 positions’
Rotate know by ‘3 positions’

Part2.2: KV-Cache

Autoregressive decode task

Suppose we continuation a sentence: "There are three……"

What should we calculate in attention?

- q vector of token in position to predict.
- All k,v vectors of former tokens: `There` `are` `three` .

Suppose we decode "kinds". What should we calculate in attention next turn?

- q vector of token in new position to predict.
- All k,v vectors of former tokens: `There` `are` `three` `kinds` .

We observe that we calculate k,v vectors of `There` `are` `three` repeatedly. Can we cache it?

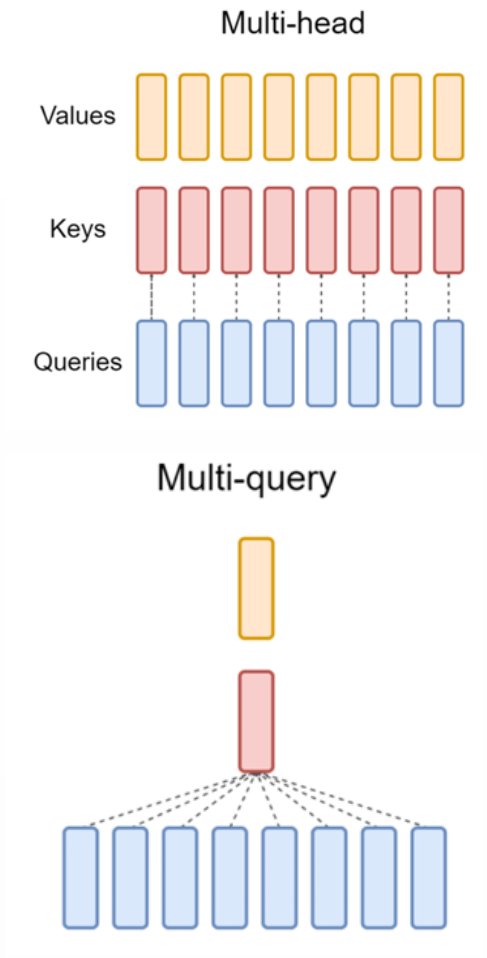
KV-Cache

We call this method "KV-Cache": Cache all former k,v vectors, so that time complexity is reduced from $\Theta(N^2)$ to $\Theta(N)$ for vectors calculation.

However, if we try to cache all vectors, it may cost an extremely large amount of memory. Can we try to optimize it, or make a compromise between accuracy and memory?

There are several methods: `MQA` `GQA` `MLA` and so on.

MQA



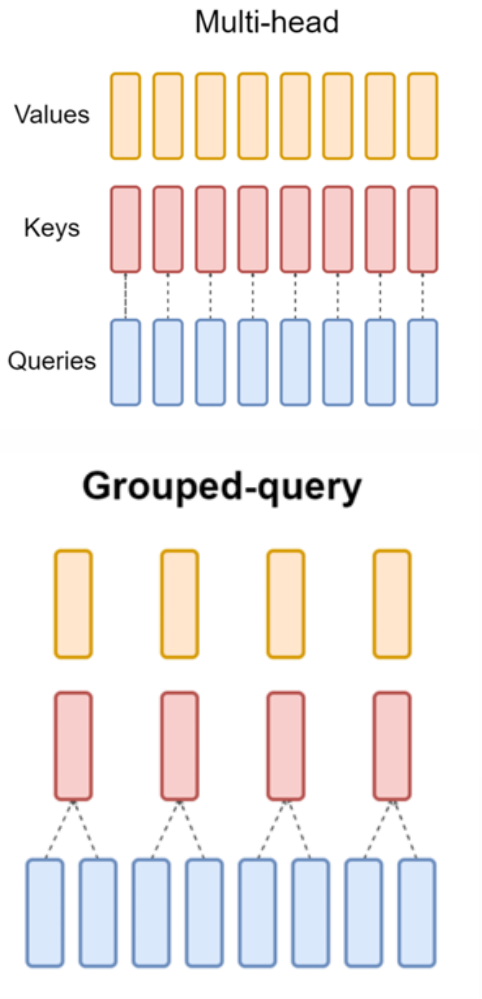
MQA (Multi-Query Attention)

- Baseline: Normal MHA.
- MQA: For multiple heads of queries, share one group of values and keys vectors.

Pros and cons?

- Greatly saves memory space.
- However, queries of different heads can only refer to one channel of information, leading to accuracy reduction.

GQA



GQA (Grouped-Query Attention)

- Baseline: Normal MHA.
- GQA: Separate all heads into G groups, each group share the same values and key vectors.

Make a balance between original MHA and MQA.

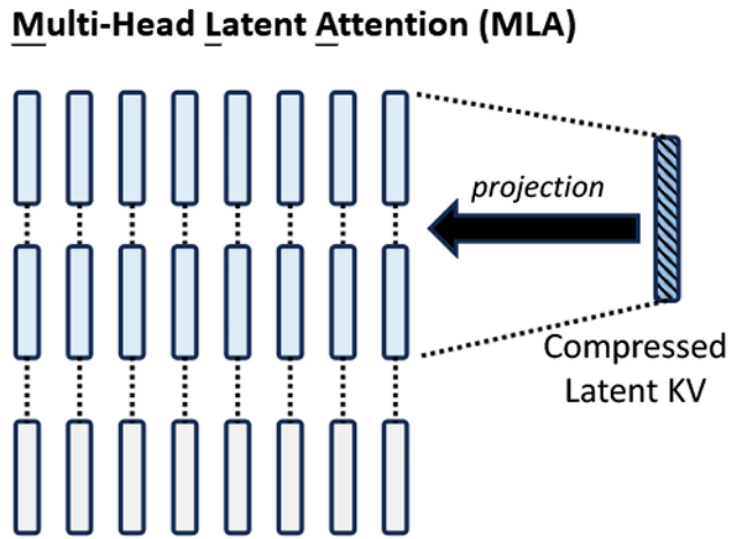
Numerous LLMs are using this method.

A new idea

- Maybe we can avoid directly store vectors, because k,v of all heads usually not full rank combining together.
- So we can try to compress k,v of all heads into a small dense vector, and try to recover k,v when needed. That is: we just need to store the small vectors, and need a little more calculation of restoration.
- This is Multi-Head Latent Attention (MHA) proposed in Deepseek-V2.

https://arxiv.org/abs/2104.09864

MLA



- Compress: $c_{KV,t} = x_t W_{DKV}$. From $R^{d_{model}}$ to R^{d_c}
- We can restore k in head h: $k_t^{(h)} = W_{UK}^h$, same for v: $v_t^{(h)} = c_{KV,t} W_{UV}^{(h)}$
- So $\text{Score}^{(h)} = q_i^{(h)} (k_j^{(h)})^T = q_i^{(h)} (c_{KV,t} W_{UK}^{(h)})^T = [q_i^{(h)} (W_{UK}^{(h)})^T] c_{KV,j}^T$, $q_i^{(h)} (W_{UK}^{(h)})^T$ is fixed for certain token, do not need to calculate for each position.
- So for score calculation, we just need to store $c_{KV,j}$.
- For $\text{output}^{(h)} = \sum_j \text{Weights}^{(h)} \cdot v_j^{(h)} = \sum_j \text{Weights}^{(h)} \cdot c_{KV,j} W_{UV}^{(h)}$
- We just need to calculate $\sum_j \text{Weights}^{(h)} \cdot c_{KV,j}$, and multiply $W_{UV}^{(h)}$ at last. Still need $c_{KV,j}$ only.

MLA

Anything wrong?

$\Rightarrow$ We haven't consider position embeddings!

Can it still solid given RoPE applied?

- $(qR_n)(kR_m)^T = qR_nR_m^TW_{UK}^Tc_{KV}^T$

We find R_nR_m is determined dynamically by position.

That is: For certain token, we cannot just calculate one vector! It is not solid for RoPE!

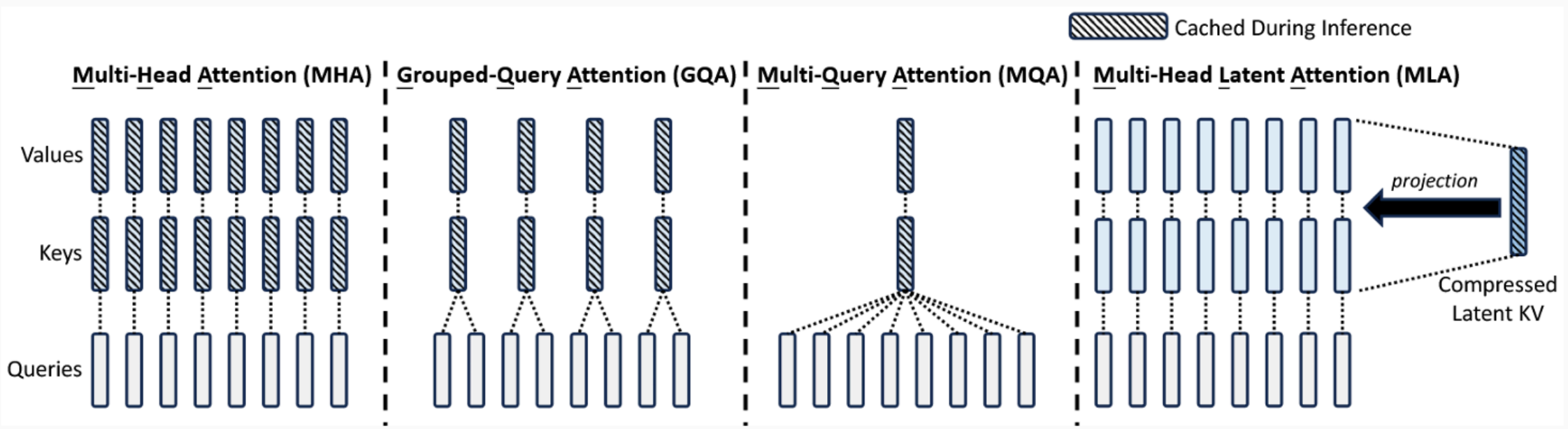
We use **Decoupled RoPE**. That is: calculate score of q, k vectors and positions separately.

- $k_{C,i}^{(h)} = c_{KV,t}W_{UK}^{(h)}$
- $k_{R,i}^{(h)} = RoPE(h_iW_{KR}^{(h)})$
- $k_i^{(h)} = [k_{C,i}^{(h)}, k_{R,i}^{(h)}]$
- $q_{C,i}^{(h)} = q_i^{(h)}$
- $q_{R,i}^{(h)} = RoPE(q_i^{(h)})$
- $q_i^{(h)} = [q_{C,i}^{(h)}, q_{R,i}^{(h)}]$

MLA

- $k_i^{(h)} = [k_{C,i}^{(h)}, k_{R,i}^{(h)}]$
- $q_i^{(h)} = [q_{C,i}^{(h)}, q_{R,i}^{(h)}]$
- $\text{Score}^{(h)} = q_i^{(h)}(k_j^{(h)})^T = q_{C,i}^{(h)}(k_{C,j}^{(h)})^T + q_{R,i}^{(h)}(k_{R,j}^{(h)})^T$
  $= q_i^{(h)}(W_{UK}^{(h)})^T c_{KV,j}^T + q_{R,i}^{(h)}(k_{R,j}^{(h)})^T$
  Term1 is the same as before, term 2 needs normal calculation of RoPE.
- $\text{output}^{(h)}$ is similar as before.

KV-Cache methods



PART2.3: Sparse attention

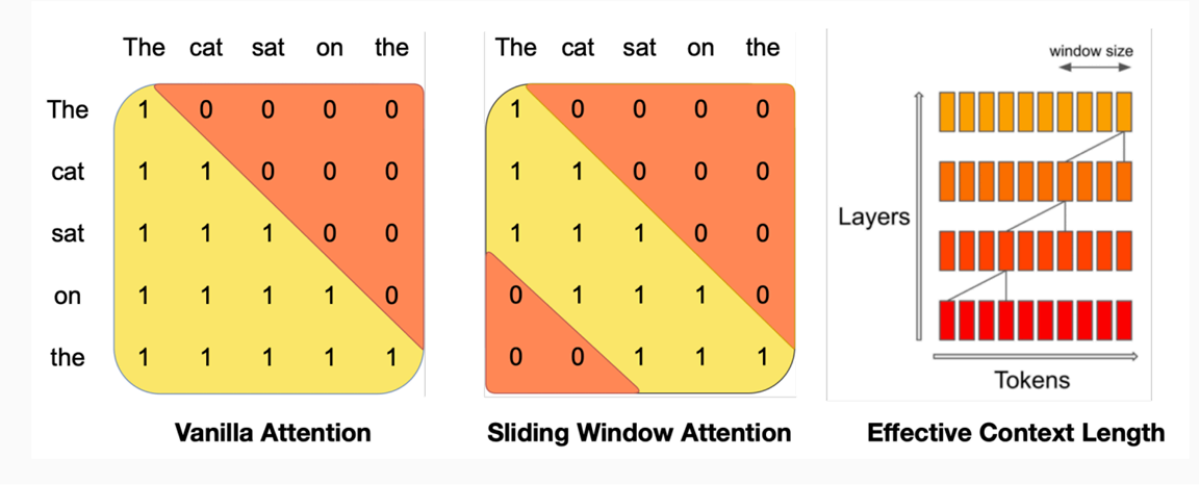
The time complexity of calculating attention is $\Theta(N^2)$. The most important cost is calculation of QK^T.

So, can we optimize the calculation method to achive a banlance between accuracy and time efficiency?

We try to calculate sparse attention. That is: only calculate a part of logits according the designed rules.

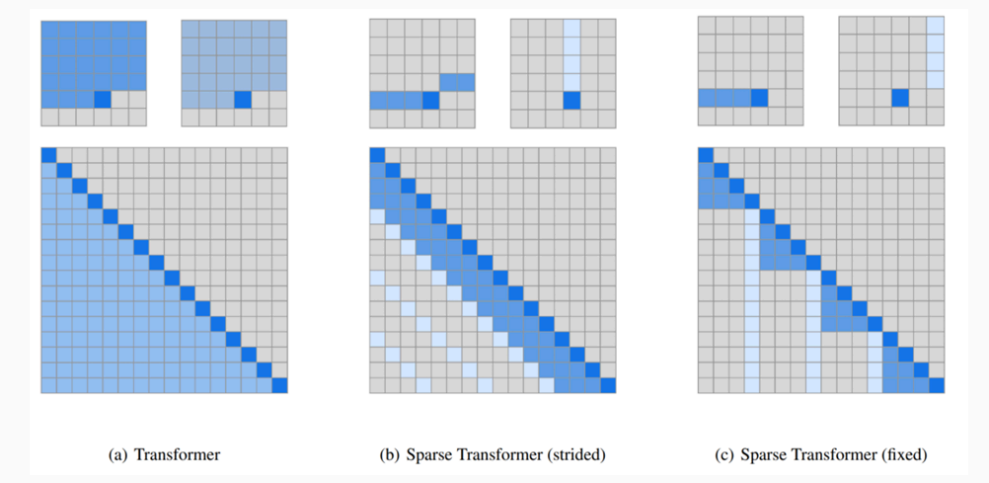
Sliding window attention

- We only "care" about the nearest k tokens. That is: we only calculate attention score in a band that is not so wide.
- Since Transformer has multilayers, information of older tokens absorbed in bottom layers will carried in tokens in the sliding window, so all tokens can absorbed all former tokens' information indirectly.



Sparse Transformer

https://arxiv.org/abs/1904.10509



- Sparse Transformer (strided)
  - Caring about recent toknes and periodic former tokens, just like "intensive reading" and "roughly review", suitable for data approximately periodic.
- Sparse Transformer (fixed)
  - Caring about modular recent tokens(such as: this line) and modular former tokens (such as: certain column), suitable for data with approximately chunk structures.

PART3: FFN

PART3.1: SwiGLU

Original Transformer FFN

$\mathrm{FFN}(x) = \sigma(xW_1)W_2, W_1 \in R^{d_{model} * 4d_{model}}, W_2 \in R^{4d_{model} * d_{model}}$

What's the activation function σ?

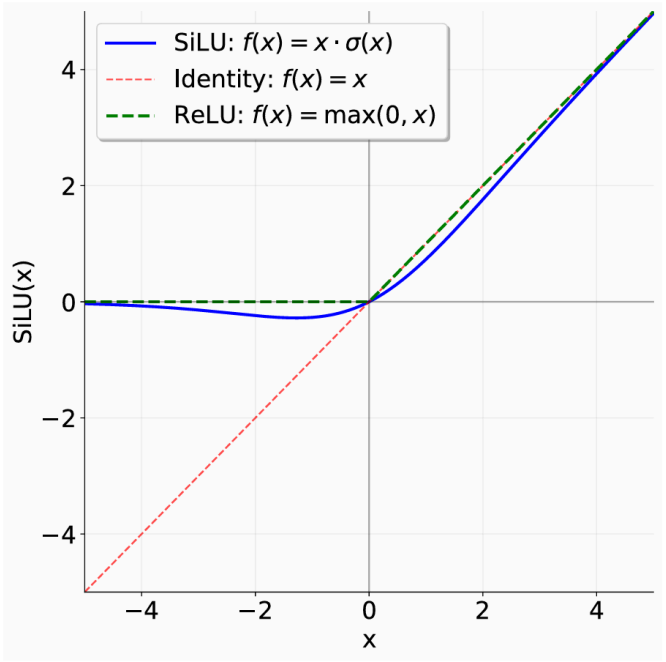
In the original transformer and other early LLMs (like T5), ReLU is often used.

$$\sigma(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

Pros and cons?

- Easy.
- But for gradient calculation:
  $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \mathrm{ReLU}(x)} \cdot \frac{\partial \mathrm{ReLU}(x)}{\partial x}$
  If $x < 0$, $\frac{\partial \mathrm{ReLU}(x)}{\partial x} = 0$, the gradient is 0! This means parameters cannot update!
  Also, if $x = 0$, it is non-differenciable.
- So it has the problem of efficient updating.

SiLU / Swish



$$\mathrm{SiLU}(x) = x \cdot \sigma(x).$$

σ is sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$

- Smooth and differenciable at $x = 0$.
- Avoid the problem of gradient $= 0$.

$$\mathrm{Swish}(x) = x \cdot \sigma(\beta x)$$

So SiLU is a special example of Swish.

All commom activation functions:

https://vitalab.github.io/blog/2024/08/20/new_activation_functions.html

Is it enough?

Forward analysis: We try to discover what's the activation elements of FFN like.

$y_i = \sigma(\sum_j x_j (W_1)_{ji})$

We find it a combination of first order interaction terms. It is hard to explicitly capture higher order interactions.

Is it enough?

Backward analysis: We try to calculate how gradients of this network change.

$y = \sigma(xW_1)$

$\Rightarrow \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = (\frac{\partial L}{\partial y} \odot \sigma'(z)) \cdot W^T$ (Tips:$z = xW_1$)

For a considerable part of activation functions, $0 < \sigma'(z) < 1$. So if $\sigma'(z) << 1$ or number of layers is large, the gradient is easy to vanish.

GLU

There's a way to solve this problem: Build another path also derive from the input x, that is a gate. Use a function σ to activate the values of gate, and do element-wise multiplication with xV.

That is:

- $y = (xV) \odot \sigma(xW_1)$ (That is: linear path and activated gate)
- $\text{output} = yW_2$

Compared with original FFN, one gate is added to dynamically control the activations of input.

We call it gate activation. This structure is called GLU.

GLU

Different activation function σ determine different kinds of GLUs.

- $\mathrm{FFN}_{\mathrm{ReGLU}}(x, W_1, V, W_2) = (\mathrm{ReLU}(xW_1) \odot xV)W_2$
- $\mathrm{FFN}_{\mathrm{GeGLU}}(x, W_1, V, W_2) = (\mathrm{GeLU}(xW_1) \odot xV)W_2$
- $\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W_1, V, W_2) = (\mathrm{Swish}(xW_1) \odot xV)W_2$

Many modern LLMs, such as LLAMA, are using SwiGLU as FFN.

So what's the advantage of them?

Forward analysis

$$y = (xV) \odot \sigma(xW_1)$$
$$\Rightarrow y_i = \mathrm{Swish}(\textstyle\sum_k x_k W_{k,i})(\sum_j x_j V_{j,i})$$

That is a combination of second order interaction terms ($c_{ij}x_i x_j$). Comparing with original FFN, it can explicitly modeling higher order interactions.

Backward analysis

$y = (xV) \odot \sigma(xW_1)$
$\Rightarrow \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}[(\sigma(z)V^T) + (xV) \odot \sigma'(z)W_1^T]$

For $\sigma(z)V^T + (xV) \odot \sigma'(z)W_1^T$:

- Term2 may also vanished because $\sigma'(z)$ may be small.
- But term 1 is depended on value of activated gate $\sigma(z)$: if the gate is open, the gradients will flows through.

So it avoid gradient vanishing, and parameters' updating is controlled by the gate.

New scale

Comparing with the original FFN, there exist a new learnable matrix (the gate).

In order to make number of parameters equals to original FFN (with $d_{ff} = 4d_{model}$), we should change d_{ff} of SwiGLU.

$d_{ff} = \frac{8}{3} d_{model}$

PART3.2: MoE

For an extremly large LLM

We want to design a model structure with huge number of parameters. What can we do?

- Of course you can scale up the original transformer. But one of the problems is: The larger model means the longer inference time. The model may be extremly slow because of the large scale matrix calculation.

There exist a new method: Mixture of Experts (MoE).

Divide the work and be experts

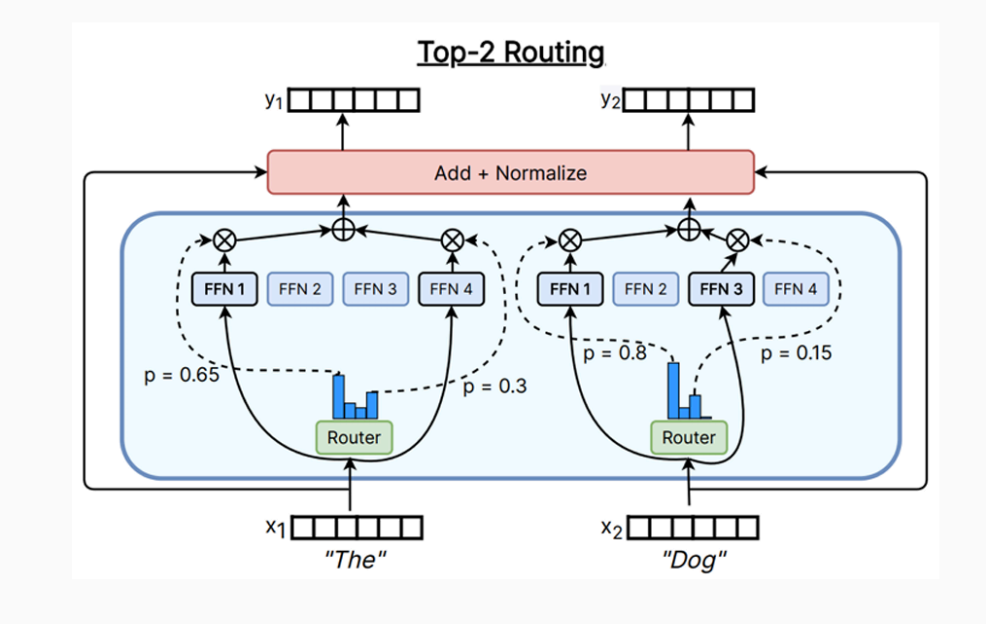
In the original Transformer, all subject of tasks share one FFN. This means the single FFN should carry all kinds of modeling works.

But in real life, some heavy works always done by division of labor: Everyone does different things and excels in their own ways.

This is mixture of experts:

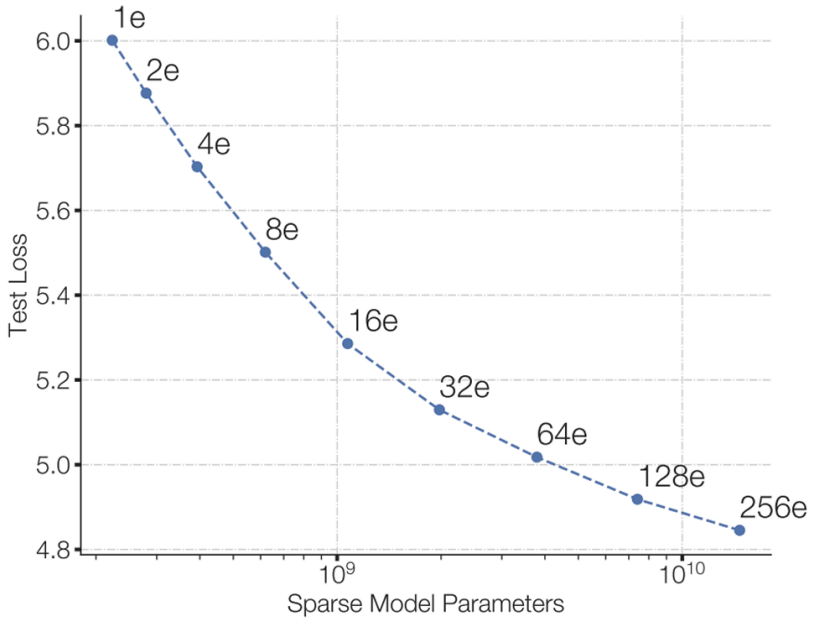
- Use multiple FFN, each FFN represents a subject.
- For each token, decide which FFN to enable.
- Only use a few, even one of them.
- The model has a huge number of parameters, but the inference speed almost unchange.

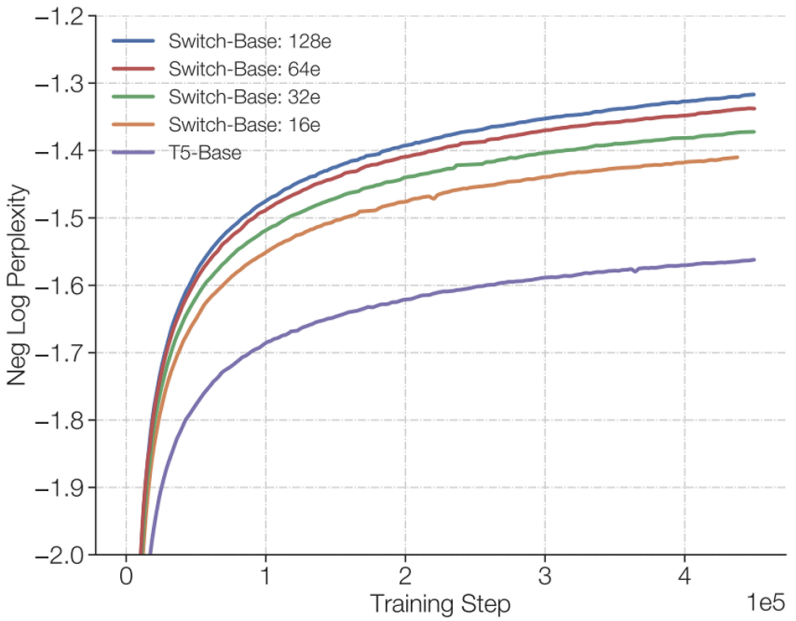
MoE



- For each token, enter router layer first.
- The router layer is usually a linear activation+softmax.
- Sample top k experts (k is very small) according to the distribution.
- For k FFN outputs, weight and accumulate them as the final output.
- Others remain the same.

More parameters and same FLOP





PART4: Normalization

PART4.1: Normalization methods

Batch Norm

We use `batch_size` samplings to run a training step. This means we need to do `batch_size` forward process in parallel. For each tensor in the embedding process, we do normalization for each element based on **`batch_size` elements at the same position**.

batch_size = 2 seq_len = 4 embed_size = 5

句子一	关	0.32	0.37	0.75	0.15	0.72
	注	0.44	0.96	0.52	0.37	0.87
	我	0.02	0.05	0.78	0.81	0.19
	PAD	0.41	0.34	0.89	0.15	0.27
句子二	爱	0.52	0.77	0.52	0.03	0.65
	有	0.95	0.71	0.15	0.96	0.88
	温	0.93	0.03	0.71	0.22	0.44
	度	0.93	0.22	0.23	0.55	0.03

Batch Norm

$$y_i = \frac{x_i + \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}\gamma_i + \beta_i$$

μ_i, σ_i are calculated from `batch_size` number of elements at the same position.

We use learnable γ and β to restore the power of representation (For example, some features should be really big to activate the network).

Batch Norm

It is used more frequently in CV, but not in LLM.

- If batch size is small, for example, equals to 1 during inference, it is very unstable.
- Cause distortion of μ, σ for sentences with varying length (if we use padding mask).
    - We will initially let the embedding of padding mask be 0 or a certain vector, and it has no actually mean, which should not be included in calculation.

Layer Norm

Compared with Batch Norm: we use `d_model` number of elements at the same tensor to calculate μ_i, σ_i. Others are the same.

$$y_i = \frac{x_i + \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}\gamma_i + \beta_i$$

Used in the original Transformer and some other LLMs, such as BERT, GPT-2.

batch_size = 2 seq_len = 4 embed_size = 5

句子一	关	0.32	0.37	0.75	0.15	0.72
	注	0.44	0.96	0.52	0.37	0.87
	我	0.02	0.05	0.78	0.81	0.19
	PAD	0.41	0.34	0.89	0.15	0.27
句子二	爱	0.52	0.77	0.52	0.03	0.65
	有	0.95	0.71	0.15	0.96	0.88
	温	0.93	0.03	0.71	0.22	0.44
	度	0.93	0.22	0.23	0.55	0.03

Root Mean Square Layer Norm (RMSNorm)

https://arxiv.org/abs/1910.07467

What we do in Layer Norm?

- Re-centering (minus μ)
- Re-scaling (divide σ)

Through experiments, it turns out that Re-centering is not so important. Since normalization module will be called multiple times in forward process, can we adjust it to save FLOPs?

Root Mean Square Layer Norm (RMSNorm)

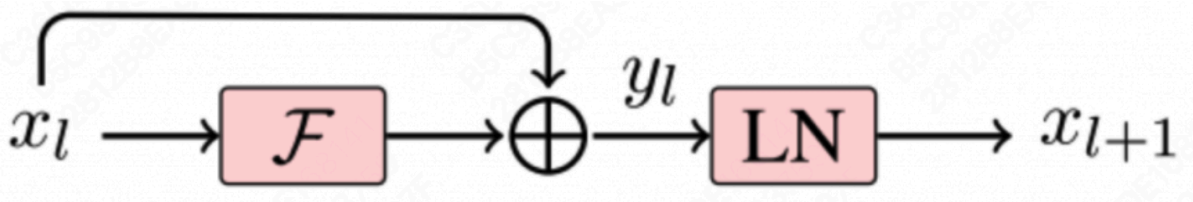
$$y_i = \frac{x_i}{\mathrm{RMS}(x)} g_i$$

$$\mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_i x_i^2 + \epsilon}$$

- We need to walk through the tensor twice to calculate μ and σ in LN, but once in RMSNorm.
- Performance almost do not declined, even better in some conditions.

PART4.2: Normalization position

Post-norm



$$x_{t+1} = \mathrm{Norm}(x_t + F_t(x_t))$$

That is: Normalization after calculating residual connection.

Used in the original transformer and BERT.

Post-norm

But actually, this normalization is not so stable during training process.

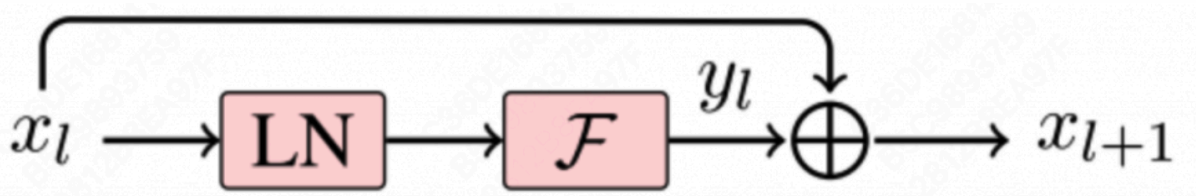
$$\frac{\partial L}{\partial x_t} = \frac{\partial L}{\partial x_{t+1}} \frac{\partial x_{t+1}}{\partial x_t} = \frac{\partial L}{\partial x_{t+1}} J_{norm,l}(I + \frac{\partial F_t}{\partial x_t})$$

This means we will multiply $J_{norm,l}$ continuously, which means the hidden danger of vanishing or exploded gradient.

Almost all LLMs now are using pre-norm.

Pre-norm

$$x_{t+1} = x_t + F_t(\mathrm{Norm}(x_t))$$



$$\frac{\partial L}{\partial x_t} = \frac{\partial L}{\partial x_{t+1}} \frac{\partial x_{t+1}}{\partial x_t} = \frac{\partial L}{\partial x_{t+1}} (I + \frac{\partial F_t}{\partial x_t}).$$

The form of this expression is similar to residual connection, and reduce the danger of gradient problems.

Summary of Transformer architecture and modules

Name	Year	Norm	Parallel Layer	Pre-norm	Position embedding	Activations	Stability tricks
Original transformer	2017	LayerNorm	Serial	☐	Sine	ReLU	
GPT	2018	LayerNorm	Serial	☐	Absolute	GeLU	
T5 (11B)	2019	RMSNorm	Serial	☑	Relative	ReLU	
GPT2	2019	LayerNorm	Serial	☑	Absolute	GeLU	
T5 (XXL 11B) v1.1	2020	RMSNorm	Serial	☑	Relative	GeGLU	
mT5	2020	RMSNorm	Serial	☑	Relative	GeGLU	
GPT3 (175B)	2020	LayerNorm	Serial	☑	Absolute	GeLU	
GPTJ	2021	LayerNorm	Parallel	☑	RoPE	GeLU	
LaMDA	2021			☑	Relative	GeGLU	
Anthropic LM (not claude)	2021			☑			
Gopher (280B)	2021	RMSNorm	Serial	☑	Relative	ReLU	
GPT-NeoX	2022	LayerNorm	Parallel	☑	RoPE	GeLU	
BLOOM (175B)	2022	LayerNorm	Serial	☑	AliBi	GeLU	
OPT (175B)	2022	LayerNorm	Serial	☑	Absolute	ReLU	
PaLM (540B)	2022	RMSNorm	Parallel	☑	RoPE	SwiGLU	Z-loss
Chinchilla	2022	RMSNorm	Serial	☑	Relative	ReLU	
Mistral (7B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
LLaMA2 (70B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
LLaMA (65B)	2023	RMSNorm	Serial	☑	RoPE	SwiGLU	
GPT4	2023			☐			
Olmo 2	2024	RMSNorm	Serial	☐	RoPE	SwiGLU	Z-loss, QK-norm
Gemma 2 (27B)	2024	RMSNorm	Serial	☑	RoPE	GeGLU	Logit soft capping, Pre+post norm
Nemotron-4 (340B)	2024	LayerNorm	Serial	☑	RoPE	SqRelu	
Qwen 2 (72b) - same for 2.5	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Falcon 2 11B	2024	LayerNorm	Parallel	☑	RoPE	GeLU	Z-loss
Phi3 (small) - same for phi4	2024	RMSNorm	Serial	☑	RoPE	GeGLU	
Llama 3 (70B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Reka Flash	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Command R+	2024	LayerNorm	Parallel	☑	RoPE	SwiGLU	
OLMo	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Qwen (14B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
DeepSeek (67B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Yi (34B)	2024	RMSNorm	Serial	☑	RoPE	SwiGLU	
Mixtral of Experts	2024			☐			
Command A	2025	LayerNorm	Parallel	☑	Hybrid (RoPE+NoPE)	SwiGLU	
Gemma 3	2025	RMSNorm	Serial	☐	RoPE	GeGLU	Pre+post norm, QK-norm
SmolLM2 (1.7B)	2025	RMSNorm	Serial	☑	RoPE	SwiGLU	

- Most use RMSNorm
- Most use Pre-Norm
- Most use RoPE
- Most use SwiGLU

Thinking

For normalization module, it seems that using `RMSNorm` *cannot save much FLOPs because matrix calculation in other modules actually cost over* 99% *FLOPs. So why do we still need* `RMSNorm` *?*

Layernorm example

```
mean = x.mean(dim=-1, keepdim=True) # Pass x for the 1st time
var = x.var(dim=-1, keepdim=True) # Pass x for the 2nd time
y = (x - mean) / sqrt(var + eps) * gamma + beta
```

- When first pass `x`, we need to load `x` from global memory to calculate `mean`.
- When pass `x` for the second time, `x` is most likely removed from cache because of the amount of resource
- Comparing with matrix calculation (with mature caching mechanism), we should frequently load from global memory, which is the main bottleneck.
- So normalization cost about 25% time consumption with only 0.17% FLOPs.